Convex Hull Approximation

Hoang Nguyen

Undergraduate

August 29, 2018

Abstract

This paper dives in to an algorithm for fast convex hulls. The paper introduces the subject of convex hulls and then implements Professors Calder and Oberman's algorithm. It will go into detail of using half-spaces to collect points as you push an amount h into the point cloud. The algorithm would then calculate the half-spaces that collected the most points and assign those as the sides of the convex hull. It will show how to implement all of this using Matlab. The objective of this algorithm is to give a optimal runtime of finding a convex hull.

1 Introduction

Computational geometry is a growing field full of various, diverse problems including convex hulls. A geometric set is convex if for every two points in the set, the line segment joining them is also in the set. One of the first problems identified in the field of computational geometry is that of computing the smallest convex shape, called the convex hull, that encloses a set of points.[GO17] It is only a convex hull if all the points on the outside of the point cloud are convex when connected and the shape encapsulates all of the points inside, or in other words, the interior angle points inward as illustrated in Figure 1. If this is not the case, the shape would not be a convex hull.

There are multiple ways to find the convex hull. Some of the most common ways are the Graham's Scan, Gift Wrapping/Jarvis March, and Chan's algorithms. While the methods for

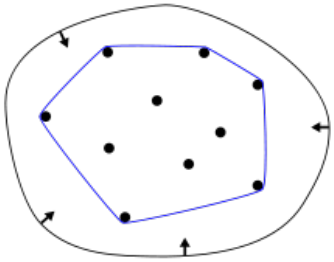

Figure 1: An example of a convex hull. Although the picture does not show, it includes the points inside the convex as well. [cona]

determination have very different techniques and principles, the primary factor that distinguishes them is their computational complexity. Computational complexity essentially refers to how fast it takes an algorithm to run. The objective is to have the fastest running time.

In this paper, we will be focusing on an algorithm that can compute a fast convex hull. What makes this algorithm faster than the popular aforementioned techniques is that it is an approximation rather than a full convex hull. The algorithm will calculate an outer and inner convex hull and use a combination of the two to get an accurate approximation, however, for larger amounts of vertices's or large dimensions, it is impractical to get an exact convex hull [Mou12]. Since the algorithm is focusing on a fast approximation and not calculating the full convex hull, then ideally the runtime will be significantly faster. With a faster runtime, calculation for mass amounts subsequently becomes more quick and efficient, and bottlenecks are effectively limited.

2 Introduction to Convex Hulls

2.1 Convex Hull Algorithms

In this section we will briefly discuss some basic convex hull algorithms to introduce the common, foundational themes and ideas for calculating a convex hull, also referred to as a convex envelope. The following definitions illustrate the aforementioned concepts:

Convexity: A set S, is convex if given any points $p, q \in S$ any convex combination of p and q is in S, or equivalently, the line segment $\overline{pq} \subseteq S$ [Mou12]

Convex Hull: The convex hull of any set S is the intersection of all convex sets that contains S, or more intuitively, the smallest convex set that contains S. This can be notated as $CH(S)$ [Mou12]

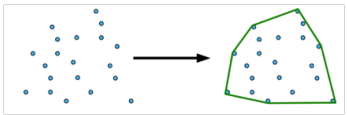

Figure 2: This is a point set with its convex hull. Although not pictured, the inner region should be shaded. [conb]

In the plane, multiple points are clustered "relatively together." For example, there are n points in P plane. The goal is to determine an output of a convex shape when iterating through the points on the outer perimeter of the internal points (called the *extreme* points) so it encapsulates all of the points inside the convex [Mou12]. Figure 2 depicts an example of this. While the orientation of iteration does not necessarily need to be ordered, counter-clockwise operations are typically

preferred as it results in more positive orientations. Of course, any direction can be translated into another, so any orientation issue can always be resolved. The output should then be a shape of the extreme points without any of the interior points. While the convex hull problem can be solved in various ways with various run times, we find the highlighted method to be the most efficient.

2.2 Gift Wrapping/Jarvis March Algorithm

The Gift Wrapping Algorithm, also commonly referred to as the Jarvis March Algorithm, is relatively simple. Starting from the most left point, the angles with the largest interior angle (with respect to said angle) are calculated. In other words, it connects to the leftmost point and finds the next point with the largest interior angle as shown in Figure 3. These calculations continue until returning to the original point, and it should have a correctly outputted convex hull. The run time for this algorithm is O(nh) since the actual calculating of the angle takes O(n) time and then it is calculated for h times, h being the vertices in the convex hull. [Mou12]

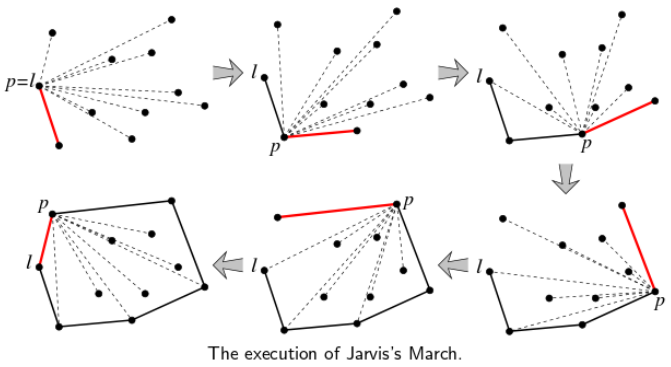

Figure 3: This is an example of the Jarvis March Algorithm [jar]

2.3 Graham's Scan Algorithm

Graham's Scan algorithm dates back to the early 1970s but is still relevant and intuitive. Beginning at the most left or bottom point, it traces its path to every point in the point cloud. Then it iterates through each point to see if the point is on the inside or the outside. If it finds that it is on the outside, it adds it to the stack and traces, and if it finds that it is on the inside, then it pops the current point off and goes back one iteration until it finds a point that is on the outside and proceeds from there. The runtime for this algorithm is O(n log n) as can be seen in Figure 4. [Mou12]

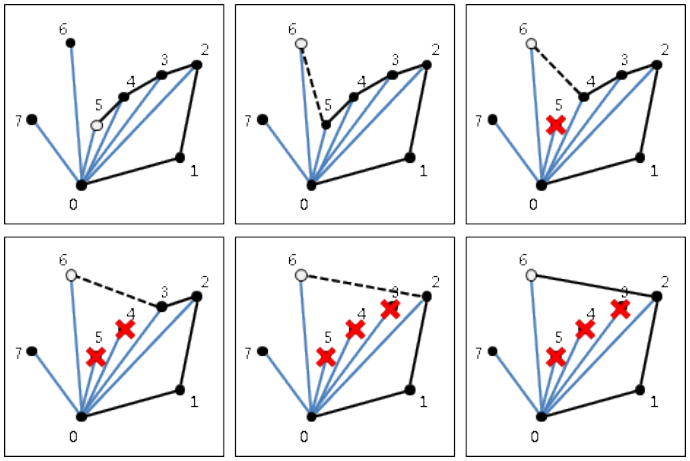

Figure 4: This is an example of the Graham's Scan. It goes from top left to right, then down to the second row left to right. [gra]

2.4 Chan's Algorithm

Chan's Algorithm takes two slower algorithms to make an algorithm that is faster than any of the two. The runtime for this algorithm is O(n log h). It is a combination of Jarvis March and Graham's Scan algorithms. Initially, it takes the point cloud and partitions it into even groups of m. Then it runs the Graham's Scan on m groups and then finishes by calculating the Jarvis March on each of those convex hulls, taking in only the extreme points of the collective group. There is one obscurity such that the value to be set m to make it efficient is essentially unknown. For this algorithm to work as intended, $h \leq m$, however, because h is unknown beforehand, to estimate a good number, you can try to iterate m = 1,2,3...,n times until the desired speed is obtained, but unfortunately the calculation is inefficient. Using a binary search would only yield in a O(n log n) time which is also not optimal. Instead, a more effective method is rapidly increase the value of m. Using the equation $m = h^c$ for some constant c, the desired runtime can be determined. The previous steps are part of a doubling search and with this method, we can maintain our desired runtime. [Mou12] An example of this is shown in Figure 5.

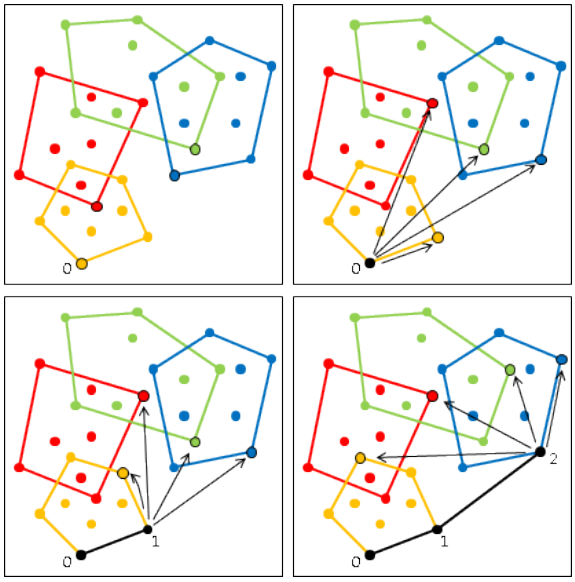

Figure 5: This is an example of Chan's Algorithm [cha]

3 Fast Approximate Convex Hull

3.1 Theory

The objective of this algorithm is to find the most efficient path to determine the approximation using half-spaces. If a point cloud in a simplex were to be taken and take the half-spaces from all angles, it would be pushed in towards the point cloud by a amount h. This can be done utilizing the unit circle. The outer convex hull can be gathered by taking the top three half-spaces that collected the most points. With this method, ideally the most amount of points along the straight sides would be collected.

You can also use this same method to calculate the interior convex hull of the simplex. Using the same strategy as before, find the half-spaces that collected the least amount of points. These areas become the outer vertices. Then connect the three extreme points to create an interior convex hull.

The approximation of the entire convex hull could then be calculated using the combination of the outer and inner convex hull of the simplex. In Oberman's version of the algorithm, he produced a similar method to calculate the ideal amount of tests for a helicopter that is now in production! [GO17]

3.2 A New Approximation

First let $\mathrm{X} \subset \mathbb{R}^d$. Then let $\mathrm{D} \subset \mathbb{S}^{d-1}$ be a finite collection of unit vectors. Then for $p \in \mathrm{D}$ let [Cal]

$$u(p) = \max_{x \in \mathrm{X}} (p \cdot x)$$

The objective is to take all the possible unit vectors and dot them with our points in the point cloud, then store the max values as the half-spaces for all the vertices. Next, we fix $h > 0$ and define for $p \in \mathrm{D}$ let [Cal]

$$N_h(p) = \#\{x \in \mathrm{X} : p \cdot x \geq u(p) - h\}$$

Now it will retrieve the dot product of p and x and store the ones that are larger than $u(p) - h$. With every h step, it will collect points along the way with each h. The objective is to be able to find the half-spaces that collect the most amount of points so the outer convex hull can be calculated.

The equations referenced as of here will determine the half spaces that touch the extreme vertices around the simplex. The issue now is that there may be some half-spaces that are on the same side of the simplex. Due to the random nature of the vertices inside the simplex, this is expected. In order to account for this, filter out the like half-spaces and keep only the unique half-spaces that were able to collect the most amount of points.

We are able to achieve this by using the following equation: Let $\epsilon > 0$ and define [3]

$$\kappa_\epsilon(p) = \sup\{h > 0 : N_h(p) \leq \epsilon n\}$$

Select some epsilon ϵ to find the appropriate amount to differentiate by. That means if point is ϵn is less or equal, it will be discarded. If the amount is more than ϵn, then the point would be kept.

These same equations can be modified slightly to help us find the interior vertices that can help us find our inner convex hull. [Cal]

$$N_h(p) = \min_{p \in \mathrm{D}} \#\{y \in \mathrm{X} : p \cdot y \geq p \cdot x - h\}$$

and

$$\kappa_\epsilon(x) = \sup\{h > \mathbb{R} : N_h(x) \leq \epsilon n\}$$

For the interior points we want to compare the dot products of the unit vectors and points to

the same minus h. This time, instead of looking for the points whose half-spaces collect the most amount of points, search for the ones that collected the least. Ideally these would be our three corners of the simplex. Then the kappa equation is very similar as well.

Now we have our equations for the inner and outer convex hulls [Cal]

$$CH_{\text{in}}(\text{X}) = \text{ConvHull}(\{x \in \text{X} : \kappa_\epsilon(x) > \Theta\})$$

and

$$CH_{\text{out}}(\text{X}) = \bigcap_{p \in \text{D}:\kappa_\epsilon(p) < \theta} (\{x \in \mathbb{R}^d : p \cdot x \leq u(p)\})$$

where θ and Θ are both positive parameters. From this point one could develop a method where both the inner and outer convex hulls could be put together to get a general convex hull for the entire simplex.

3.3 Implementation

To have a visual example, we have built these algorithms in Matlab to test. We will go over portions of the program to clarify the logic. A complete version of the code can be found later in this section.

```
n = 1000;
h = 0.1;
e = 0.1;
```

Here we set up our variables. n represents the number of points we will sample and h and e are our steps and variance respectively.

```
points = rand(n,2);
theta = 0:0.01:2*pi;
p = [cos(theta'),sin(theta')];
O = points(:,2) - 2*points(:,1) > 0;
points(O,:) = [];
O = points(:,2) + 2*points(:,1) > 2;
points(O,:) = [];
hull = scatter(points(:,1),points(:,2),'yellow');
hold on;
```

These lines set up our simplex. Using n, we create 1,000 random points that fill the simplex. We also initialize our p which is all the values in the unit circle. Then we plot the points into our graph.

```
function [ value ] = u( p,points )
```

```
for i = 1:length(p)
    a = -Inf;
    for j = 1:length(points)
        a = max(a,p(i,1)*points(j,1)+p(i,2)*points(j,2));
    end
    value(i) = a;
end
```

This is our code for the $u(p)$ function. It takes in two arguments, p and points. The function loops through both of them to get the dot products for our half-spaces. We set a to an arbitrary negative number, so then in our loop, when a finds a max, it stores that value along with the index into the variable *values*. This information is later stored into a variable in the main program called $max(U)$.

```
function [ result, counter] = Nh( p,points,arg3)
result = zeros(length(p),1);
counter = zeros(length(p),1);
for i = 1:length(p)
    for j = 1:length(points)
        pxDot = (p(i,1)*points(j,1)+p(i,2)*points(j,2));
        if pxDot >= arg3(i)
            result(i) = j;
            counter(i) = counter(i) + 1;
        end
    end
end
```

This is the code for the outer $N_h(p)$ function. We take in three arguments: p, points, and arg3 which is just the variable $maxU - h$. We then loop through, calculating the dot products of p and points, however, this time, we check the condition that if the value of that dot product is larger or equal to $maxU$, then we would store that into the variable result. Subsequently we also record the index in another variable called counter. As a result, we will be able to sort these in descending order so vertices with the half-spaces that collect the most points appear at the top of the stack. This is accomplished using the following code:

```
[g,I] = sort(index, 'descend')
```

Next we want to modify the code a little bit so it can find the vertices that collect the least amount of points. (e.g. For the equation $N_h(p) = \min_{p \in \mathrm{D}} \#\{y \in \mathrm{X} : p \cdot y \geq p \cdot x - h\}$.

```
N = zeros(length(points),1);
ph = zeros(length(points),1);
for k = 1:length(points)
    pcount = zeros(length(p),1);
```

```
    a = Inf;
    for i = 1:length(p)
        pxDot = p(i,1)*points(k,1)+p(i,2)*points(k,2);
        for j = 1:length(points)
            pyDot = p(i,1)*points(j,1)+p(i,2)*points(j,2);
            if pyDot >= pxDot - h
                pcount(i) = pcount(i) + 1;
            end
        end
    end
    [a, i] = min(pcount);
    N(k) = a;
    ph(k) = i;
end
```

This code block is very similar to the one that calculates for the outer convex hull. Instead, we are now comparing the dot products of p and points to the value of the dot products - h. Only then do we store the amounts into the variables $N(k)$ and $ph(k)$.

We then sort these values as well using *sort*. Matlab automatically sorts by ascending order, so we can execute the following:

```
[Nhs, J] = sort(N)
J(1:20)
```

Now our lists are sorted and we are only checking the first twenty items as specified by $J(1:20)$.

```
dtheta = 0.5;
thetaList = zeros(0);
x = -0.1:0.01:1.1;
j = 1;
while length(thetaList) <= 2
    y = (maxU(I(j)) - p(I(j),1)*x) / p(I(j),2);
    ans = theta(I(j));
    a = 1;
    for k = 1:length(thetaList)
        if abs(ans-thetaList(k)) <= dtheta
            a = 0;
        end
    end
    if a == 1
        pause;
        plot(x,y,'blue')
        thetaList(end+1) = ans;
    end
    j = j + 1;
end
```

The above code is for our κ function, but it is slightly modified for our purposes. Now instead of ϵ, we use the variable dtheta which are essentially identical in nature. We initialize a list called thetaList and then check our half-spaces.

The loop will automatically draw our first line. Then for the next iteration, it will compare the first line to the new line. If the line is near the value of dtheta, it will mark our variable a (which is used as a Boolean) to zero and iterate again. It will continue until we get three unique half-spaces. By checking the length of our thetaList variable where the half-spaces are stored, the program can determine the half-spaces that form the convex hull.

The function to filter our corner vertices is almost identical.

```
pointsList = zeros(0);
i = 1;
interior = zeros(3,1);
while length(pointsList) <= 2
    ans1 = points(I(i));
    a = 1;
    for j = 1:length(pointsList)
        if abs(ans1-pointsList(j)) <= dtheta
            a = 0;
        end
    end
    if a == 1
        scatter(points(J(i),1),points(J(i),2),'red')
        pointsList(end+1) = ans1;
        interior(i,1) = points(J(i),1);
        interior(i,2) = points(J(i),2);
    end
    for z = 1:length(interior) %Trying to isolate 3 unique points.
        if points(z,1) == 0
            points(z,1) = [];
        end
        if points(z,2) == 0
            points(z,2) = [];
        end
    end
    i = i+1;
end
hold off;
```

The above portion of code functions similarly to the function before, but instead of storing the half-spaces and drawing them, this function highlights our corner vertices. The primary difference is the last bit of code that starts with $forz = 1 : length(interior)$...., and this distinction will be addressed in section 4.2.

The next step would be to implement Oberman's algorithm to combine them both for a better approximation of our inner and outer convex hulls.

Here we will list the code in its entirety without any breaks:

This is the code for maxU:

```
function [ value ] = u( p,points )
%  u(p) = max(dot(p,x)) with x element of X

for i = 1:length(p)
    a = -Inf;
    for j = 1:length(points)
        a = max(a,p(i,1)*points(j,1)+p(i,2)*points(j,2));
    end
    value(i) = a;
end
```

This is the code for $N_h(p)$:

```
function [ result, counter] = Nh( p,points,arg3)
%   This function will tell you how many points are collected in the point
%   cloud for every h step
result = zeros(length(p),1);
counter = zeros(length(p),1);
for i = 1:length(p)
    for j = 1:length(points)
        pxDot = (p(i,1)*points(j,1)+p(i,2)*points(j,2));
        if pxDot >= arg3(i)
            result(i) = j;
            counter(i) = counter(i) + 1;
        end
    end
end
```

This is the main code:

```
n = 1000;

%initialize variables needed
h = 0.1;
e = 0.1;

%Take the input value and create input amount of points
points = rand(n,2);
%create all the points within the unit circle
theta = 0:0.01:2*pi;
p = [cos(theta'),sin(theta')];
```

```
%Take only half the points in the heap and keep the points that are less
%than 1 to create a simplex
O = points(:,2) - 2*points(:,1) > 0;
points(O,:) = [];
O = points(:,2) + 2*points(:,1) > 2;
points(O,:) = [];
%draw the points
hull = scatter(points(:,1),points(:,2),'yellow');
hold on;

%Having the graph instantiated, we will dot p and all the points in the
%simplex. Function takes 2 matrixes.
maxU = u(p,points);

%Next we want to determine the approximate half spaces of the simplex. This
%function will take in 3 arguements: p, points, and maxU-h.
arg3 = maxU-h;
[result,  index] = Nh(p,points,arg3)

%Then we will sort the indexes in descending order
[g,I] = sort(index, 'descend')

N = zeros(length(points),1);
ph = zeros(length(points),1);
for k = 1:length(points)
    pcount = zeros(length(p),1);
    a = Inf;
    for i = 1:length(p)
        pxDot = p(i,1)*points(k,1)+p(i,2)*points(k,2);
        for j = 1:length(points)
            pyDot = p(i,1)*points(j,1)+p(i,2)*points(j,2);
            if pyDot >= pxDot - h
                pcount(i) = pcount(i) + 1;
            end
        end
    end
    [a, i] = min(pcount);
    N(k) = a;
    ph(k) = i;
end

[Nhs, J] = sort(N)
J(1:20)

```

```
dtheta = 0.5;
thetaList = zeros(0);
x = -0.1:0.01:1.1;
j = 1;
while length(thetaList) <= 2
    y = (maxU(I(j)) - p(I(j),1)*x) / p(I(j),2);
    ans = theta(I(j));
    a = 1;
    for k = 1:length(thetaList)
        if abs(ans-thetaList(k)) <= dtheta
            a = 0;
        end
    end
    if a == 1
        pause;
        plot(x,y,'blue')
        thetaList(end+1) = ans;
    end
    j = j + 1;
end
pointsList = zeros(0);

i = 1;
interior = zeros(3,1);
while length(pointsList) <= 2
    ans1 = points(I(i));
    a = 1;
    for j = 1:length(pointsList)
        if abs(ans1-pointsList(j)) <= dtheta
            a = 0;
        end
    end
    if a == 1
        scatter(points(J(i),1),points(J(i),2),'red')
        pointsList(end+1) = ans1;
        interior(i,1) = points(J(i),1);
        interior(i,2) = points(J(i),2);
    end
    for z = 1:length(interior) %Trying to isolate 3 unique points.
        if points(z,1) == 0
            points(z,1) = [];
        end
        if points(z,2) == 0
            points(z,2) = [];
```

```
100             end
101         end
102         i = i+1;
103     end
104
105     hold off;
```

4 Conclusion

4.1 Final Thoughts

Overall this is a fantastic way to quickly calculate the approximation of a convex hull, but it is still currently just preliminary work. While additions will certainly enhance the elegance of the efficiency, the algorithm will not suffer typical performance hindrances should the work space extend into multiple dimensions or if the amount of data increased. We saw that algorithms like the Jarvis March, Graham's Scan, and Chan's Algorithm are just some of the methods to compute a convex hull. While this specific field of study is still growing, this algorithm seems to be a very efficient way of approximating the convex hull.

4.2 Errors

While the code is somewhat complete, it can always be improved. It can be vectorized by order of magnitude so that performance increases [**?**]. There are several odd spots that were not able to be resolved. For the most part, the half-spaces were always drawn out correctly. Once in a while, two half-spaces in the same side would be drawn, but this was rare.

The issue with the current code is that the inner vertices are being incorrectly highlighted. It works for most occurrences, but we are led to believe it depends on the random distribution of points given at the start of the program. Subsequently we tried to store it into another variable with just the vertices, but somehow points with the value (0,0) were being stored. When we checked the length of the pointsList variable at the end, it was expected to be three. Sometimes it appeared as eight as a result of (0,0) vectors somehow being recorded

The other issue is that the three verticies are not always in the correct spots. More often than not, the program returned points near the same corner. I predict this might be due to the fact that dtheta is not large enough, or there is an issue with the amount of points collected with the half-space in the missing corner. You can see an example of this below in Figure 6.

I predict that there are vectors in the simplex that are causing a bug in the code. I tried to troubleshoot the issue but came out with no success as of now.

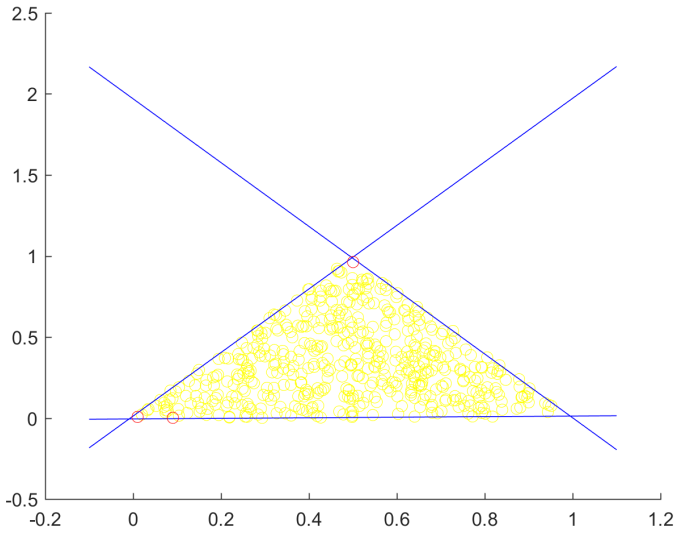

Figure 6: You can see that two verticies were marked in the bottom left corner and one marked on the right.

If I were to do recreate and retest the algorithm, I would try to see if this problem can be troubleshooted. If we can get just three vertices to be recorded, we could potentially plot a line between them all to get an inner triangle as well as already having the outer triangle.

This algorithm could also be made to extend to other dimensions other than this 2-D version that we have been working with. The goal is that this algorithm can be used up to infinite dimensions.

References

[Cal] Jeff Calder. Approximate convex hull.

[cha] *Figure 5*. Photo found at /urlhttp://www.csie.ntnu.edu.tw/ u91029/Chan'sAlgorithm1.png.

[cona] *Figure 1 picture*. Photo found at http://axon.cs.byu.edu/Dan/312/projects/project2_files/image002.png.

[conb] *Figure 2 picture*. Found at https://upload.wikimedia.org/wikipedia/commons/thumb/d/de/ConvexHull.svg/220px-ConvexHull.svg.png.

[GO17] Robert Graham and Adam M Oberman. Approximate convex hulls: sketching the convex hull using curvature. *Approximate convex hulls: sketching the convex hull using curvature*, page 1–14, Jun 2017.

[gra] *Figure 4 picture*. Found at http://www.csie.ntnu.edu.tw/~u91029/Graham'sScan5.png.

[jar] *Figure 3 picture*. Found at http://www.geeksforgeeks.org/wp-content/uploads/jarvis.png.

[Mou12] David M Mount. Cmsc 754 computational geometry, 2012.